SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 26(2), 177–194 (FEBRUARY 1996)

Compacting Garbage Collection can be Fast and Simple

CHARLES L. A. CLARKE AND DAVID V. MASON
Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada
(claclark@plg.uwaterloo.ca)
(dmason@plg.uwaterloo.ca)

SUMMARY

Copying garbage collectors are now standard for the memory-management subsystems of functional and object-oriented programming languages. Compacting garbage collection has correspondingly fallen out of favor. We revitalize the case for compaction by demonstrating that a simple compacting collector, extended with the generational garbage collection heuristic, exhibits performance as effectively as or better than a well-designed generational copying collector on real programs running in real environments. The observation that compaction preserves allocation order across collections leads to a new generalization of the generational heuristic that reduces the movement of long-lived objects. We measure the effect of substituting our compacting generational collector for a copying collector in Standard ML of New Jersey.

KEY WORDS: storage management; garbage collection; virtual memory; compaction

INTRODUCTION

The effective and efficient management of memory remains a significant challenge to the developer of a complex software system. A common solution to the problem is to isolate memory management in a single subsystem that presents an abstraction of unlimited memory. Software in the remainder of the system accesses this subsystem to allocate memory dynamically and does not explicitly deallocate the memory when it is no longer required. Instead, the memory-management subsystem periodically executes a procedure that finds memory that is no longer in use and reclaims it, making it available for future allocation. This process is termed 'garbage collection'.

Garbage collection is a heavily studied subject. Comprehensive surveys have been written by Cohen,[1] Appel,[2] and Wilson.[3] General-purpose garbage collection algorithms may be roughly divided into two categories: those that move data and those that do not. Garbage collectors in the first category move live objects (containing data still in use) together into a contiguous range, creating a large block of free memory for future allocation. Garbage collectors in the second category link garbage objects (containing data no longer in use) into a data structure from which allocation requests are satisfied. Systems that manage dynamic memory using a collector in the first category gain the advantage of extremely efficient allocation. Free memory is in a contiguous block, and allocation typically consists of little more than a pointer adjustment and a bounds test. Often, virtual memory can be enlisted to perform the bounds test.[4,5]

Garbage collectors that move data can be further divided into *compacting collectors* and *copying collectors*. These names only weakly connote the differences between the two

CCC 0038–0644/96/020177–18

Received 16 March 1995
Revised 26 June 1995

approaches. Compacting collectors are alternately termed 'in-place compacting collectors', 'sliding collectors', or 'mark-and-compact collectors'. We use 'compacting' for brevity.

Compacting collectors form a subclass of *mark-and-sweep* collectors, a large class of collectors that includes many that do not move data. Mark-and-sweep garbage collectors operate in two phases:

(1) A *mark phase* in which live data is traversed and all reachable objects are marked as live.

(2) A *sweep phase* in which one or more linear passes are made through the memory under management to separate garbage from live objects.

During the sweep phase of a compacting collector, live objects are slid toward one end of memory. The relative order of the live objects is not changed. The compacting process merely makes the live objects adjacent so that garbage is eliminated between them. Most algorithms for compacting collection were developed at a time when memory resources were scarce and therefore tend to trade off execution time for decreased memory usage,[6] placing emphasis on the minimal use of storage beyond that required for the heap. These algorithms require a small amount of fixed memory overhead per object allocated and a bounded amount of additional storage during a collection. However, the sweep phase requires two or more fairly complex passes over memory. Compacting collectors are often viewed as time inefficient and consequently have fallen into disuse in the past dozen years.

Copying collectors split the region of memory being managed into two equal *semi-spaces*. Allocations are made from one semi-space; the other is left fallow. During a garbage collection, a copying collector traverses live data, appending each live object visited to the end of a growing block at one end of the fallow semi-space. As objects are copied, forwarding pointers are left behind to indicate where the objects have been moved. These pointers are used to adjust references to moved objects. The algorithm of Fenichel and Yochelson[7] uses a depth-first traversal of the live data. The algorithm of Cheney[8] uses a breath-first traversal. Of the two, Cheney's algorithm is generally preferred as it cleverly uses the fallow semispace as a queue to guide the breath-first traversal; the algorithm of Fenichel and Yochelson requires an auxiliary stack to guide its depth-first traversal. Since only live data is touched during a garbage collection, copying collectors are often favored over mark-and-sweep collectors in general, and over compacting collectors in particular. Mark-and-sweep collectors are commonly used only when, for various reasons, objects cannot be moved.[9] Cheney's algorithm has become standard for implementing garbage collection and is widely used in practice.[2, 10]

The purpose of this paper is to rehabilitate compacting collectors as a modern memory-management method. We present a simple compacting collection algorithm that, with the exception of degenerate circumstances, requires only a single pass over memory during the sweep phase. During a collection the algorithm requires memory resources proportional to the size of the memory space being garbage collected (but there is precedent for this usage in the semi-spaces of copying collectors). The collection algorithm is amenable to augmentation with the generational garbage collection heuristic.[11] A generational version of the algorithm has been implemented for Standard ML of New Jersey (SML-NJ). We present a performance evaluation of this algorithm based on standard benchmarks. This experience indicates that a compacting collector can perform as well or better than a copying collector on real programs running in real environments. An important property of the collector is that it maintains objects in the order that they were allocated. This property improves spatial locality, which in turn improves paging and caching performance. Preserving allocation order also allows a more general approach to be taken to generational garbage collection

and can simplify the overall memory-management strategy.

A few notes on terminology: we follow the practice of referring to the client of the dynamic memory-management system as the *mutator*. We use the term *heap* to refer to the entire region of memory managed by the dynamic memory-management system. A specific region of memory being garbage collected is referred to as an *arena*. The *root pointers* or *roots* are the pointers outside the heap, typically processor registers and static storage, through which all useful heap data can be accessed, directly or indirectly. All other heap storage is considered garbage.

MOTIVATION

Our work is motivated by three assumptions that have been borne out in practice:[3, 5,11–13]

1. Almost all references between objects are from newer objects to older objects. The *ordering assumption.*
2. The longer an object has existed, the longer it is likely to continue to be useful. The *persistence assumption.*
3. Objects tend to reference other objects allocated at approximately the same time. The *locality assumption.*

Preserving allocation-order

When a copying or compacting collector is used, allocation may be implemented with a pointer update and a bounds check. This simplicity is a consequence of all available memory being in a single contiguous block. One result of this allocation method is a simple correspondence between the order in which objects appear in memory and the time at which they were allocated. Compacting collectors preserve this allocation ordering across a garbage collection, a property not shared by copying collectors.

The ordering assumption implies that if memory is maintained in allocation order (the oldest data at the lowest addresses) then there will be few pointers from lower addresses to higher addresses, and hence the accessibility graph for the heap will flow primarily down from the roots. The ordering assumption is especially relevant if the assignment operator is not provided to the mutator (e.g. the mutator is written in a pure functional programming language).

The persistence assumption implies that older objects are much less likely to become garbage in the near future than younger objects. The implication of this observation is that if we maintain the heap in allocation order, little garbage will appear in the oldest parts of the heap and we may not have to move as many objects around in that area.

Software systems that manage memory using copying garbage collectors often exhibit poor locality. That is, groups of consecutive memory references are often to addresses spread widely through memory. Poor locality results in high page fault rates and low cache hit ratios, particularly for small caches. There have been several attempts to find better ways to traverse the accessibility graph in order to improve locality.[14–16] The locality assumption implies that maintaining the heap in allocation order will improve locality.

The use of a garbage collector that preserves allocation order creates the invariant that objects do not move relative to each other unless one or more objects between them become garbage. It is possible to exploit this invariant if the mutator creates objects in groups that are guaranteed to become garbage all at the same time. Relative pointers could be used between

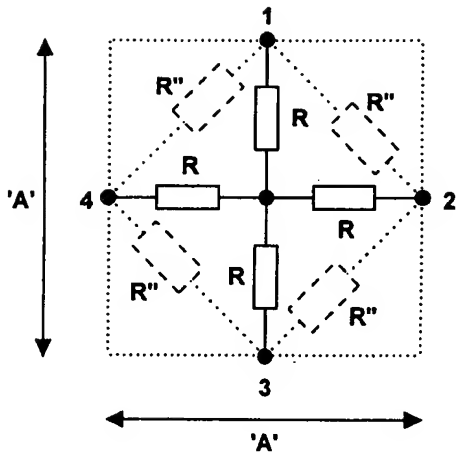

Figure 1. Generational garbage collection

objects within the group. These relative pointers indicate the offset of an object from the address at which the pointer is stored. These pointers would not need to be considered during the garbage collection process.

Generational garbage collection

Generational garbage collection is based on the ordering assumption and the persistence assumption.[11] The arena is split into two or more *generations*. Each generation has an age associated with it – objects in an older generation were allocated before those in a younger generation. The ordering assumption implies that there will be few, if any, pointers from older generations into younger generations (figure 1). When the youngest generation fills, garbage collection promotes live objects from the youngest generation into the next oldest generation. Collection on the older generation is not performed unless it too is full. This process may continue backwards through the generations until the oldest generation is reached. If the oldest generation is full, the entire heap is collected.

Garbage collection will be performed many times on a particular generation before the next oldest generation is collected. During a collection of a younger generation, objects

in older generations do not need to be traced or collected. By restricting collection to smaller portions of the heap, generational techniques significantly improve garbage collector performance.

The major confound of generational techniques is the existence of *wrong-way* pointers, pointers from older objects to younger objects. Several techniques may be used to deal with wrong-way pointers.[3] One such technique is the use of a *store list.* We note that wrong-way pointers can only be created through an assignment to a pointer variable. Each time a pointer assignment occurs, an element is added to the end of the store list recording the target of the assignment. During a garbage collection, the store list is checked for inter-generational references. These references must be considered as part of the root set when garbage collecting the younger generation. Elements of the store list that are not needed for future collections are discarded. While the use of a store list is a potentially heavy burden to place on a program written in an imperative language where pointer assignment may be quite common, it is not a major burden under a functional programming model where assignment may be rarely used.

SIMPLE COMPACTING GARBAGE COLLECTION

Our garbage collector proceeds in three phases:

1. *Mark Phase*
   A depth- or breadth-first search starting at the roots is used to find and mark accessible objects. Mark bits may be stored in the objects themselves or in a separate bitmap.
2. *Compact Phase*
   The arena is swept in allocation order. Objects are moved down to fill locations occupied by garbage. As objects are moved their new locations are recorded in a parallel *forwarding array* and references to already-moved objects are updated to reflect the new values (figure 2). The ordering assumption predicts that most references in an object being moved will be to objects that have already been moved and consequently it will be possible to adjust nearly all references during this phase. The locations of references that cannot be adjusted are recorded in an auxiliary *wrong-way pointer list.* Alternatively, the equivalent information may be extracted from a store list if one is available.
3. *Repair Phase*
   The root pointers and elements in the wrong-way pointer list are adjusted using the forwarding array. The forwarding array and wrong-way pointer list may now be discarded.

Generational collection and the density heuristic

The algorithm easily adapts to generational collection. The generations are arranged naturally in the same order as objects are allocated. Objects are compacted as they are promoted from one generation to the next.

In extending the algorithm to include the generational heuristic, we can take better advantage of the ordering and persistence assumptions than generational schemes that do not preserve allocation order. Generations are determined dynamically at any boundary between objects. This property eliminates the requirement for *a priori* decisions about which objects will be in which generations. In order to allow a generation to be declared at an arbitrary

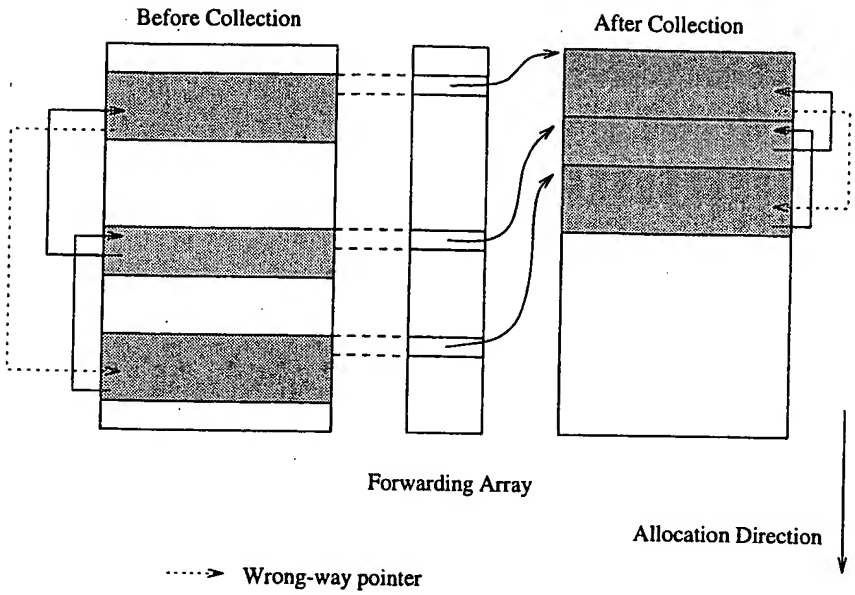

Figure 2. Illustration of the compaction algorithm

boundary, the wrong-way pointer list must be maintained between collections, since the compact phase will not examine all of the arena and consequently a wrong-way pointer list cannot be built from scratch.

After completing the mark phase, we have available information on the ratio of objects to garbage, which we refer to as *density*. This density information can be used to make a determination of whether it is worth moving objects in the oldest generation being collected. For example, if a 1MB block of objects is preceded by two words of garbage, it is clearly not worth moving the block to recover the two words of garbage.

AN IMPLEMENTATION FOR STANDARD ML OF NEW JERSEY

To test the utility of our collection algorithm and to verify the ideas we presented as design motivation, we implemented a compacting garbage collector for Standard ML of New Jersey (SML-NJ).[17,18] The collector was implemented for version 0·75 of SML-NJ. The two-generation copying collector used in that version of SML-NJ is essentially identical to that used in the newest 'official' release (version 0·93). More recent 'un-official' releases of SML-NJ use a new multi-generational copying collector that is somewhat different from the collector we replaced. However, nothing in principle would prevent us from replacing this new collector with a compacting collector based on our techniques.

The original SML-NJ collector

The run-time and garbage collector for Standard ML of New Jersey has been described in detail elsewhere.[19] We provide a brief overview here.

The garbage collector for SML-NJ is a generational garbage collector with two generations. A *minor collection* copies live objects in the younger generation into the older

generation using Cheney's algorithm.[8] A *major collection* compacts the entire heap. As the first step in a major collection, all live objects are copied to a scratch arena using Cheney's algorithm. After correcting pointers, this block of live data is moved back to the original arena, making a major collection effectively an in-place operation.

All objects start with a 4 byte descriptor that contains a 5 bit tag indicating the type of the object and a 27 bit field specifying the length of the object. A 4 byte word is the basic unit of allocation. Pointers and integers are both one word in size. Integers are distinguished from pointers by the least significant bit of the word; this bit is zero for pointers and one for integers.

Pointers normally point to the second word of an object – the word after the descriptor – but pointers to the interior of objects are possible. These *interior pointers* arise from the need for objects, especially code strings, to refer to each other using relative addressing. Such objects must remain in the same locations relative to each other, and so must be considered as parts of a larger composite object by a copying collector. A protocol is used by the garbage collector to identify interior pointers and to find the descriptor of the associated composite object. This protocol includes the use of specially tagged descriptors embedded in the composite objects.

The SML-NJ run-time system uses a store list to identify pointers from the older generation into the younger generation. Each execution of an assignment results in the addition of a node to the store list. Each store list node is a heap allocated object that points to the destination of the assignment. The store list is passed to the garbage collector at the start of each minor collection and is discarded after the minor collection.

The new collector

Our garbage collector uses the same basic scheme – two generations with a major collection coalescing the heap in place. We maintain the existing scheme for increasing and decreasing the total heap size. Apart from the garbage collector, there is only one small change in the run-time system, needed to enforce an additional mutator invariant. This invariant simplifies the maintenance of the wrong-way pointer list by permitting objects containing wrong-way pointers to be tagged as such. The mutator is not required to identify objects containing wrong-way pointers or to set the tag. These functions are performed by the garbage collector during the first collection after a wrong-way pointer is written into an object. The only burden placed on the mutator is to treat these wrong-way objects as potential targets for an assignment statement. Handling this burden affects only two lines of code in the run-time polymorphic equals function. We made no changes to the compiler itself.

A bitmap is used to mark objects. For minor collections, our original plan was to place mark bits in the objects themselves, but initial experimentation revealed that some objects – presumably known by the mutator to become garbage before the next collection – were being allocated without descriptors. Since there is no way to distinguish the descriptor of a valid object from a random word of a descriptorless object, sweeping the heap is impossible without the aid of an auxiliary data structure. For major collections, the combination of interior pointers and the details of our in-place compacting algorithm make marking directly in the objects impossible. Compacting in place overwrites the information needed to handle interior pointers. For both major and minor collections, we facilitate the adjustment of interior pointers by recording their locations in the bitmap.

During a minor collection, objects are compacted into a non-overlapping area, and for-

warding pointers are stored at the vacated locations, rather than in a separate forwarding array. During a major collection, compaction is done in place, and a parallel forwarding array is temporarily allocated in an unused portion of the heap to record forwarding offsets (the number of bytes the object was moved). Because the minimum object size is two words, one forwarding offset can serve two locations and the forwarding array needs to be only half the size of the arena being compacted. During both major and minor collections we simplify pointer adjustment by leaving forwarding information for interior pointers as well as for pointers to the beginning of objects.

During a minor collection the store list serves the same purpose as in the original collector; it contains a list of objects in the older generation that point to objects in the younger generation. In the original collector the store list is discarded at the end of a minor garbage collection. In our collector, store list elements form the basis of a wrong-way pointer list. After the mark phase of each minor collection, the store list is scanned and elements that point to wrong-way pointers in marked objects are themselves marked and placed in a separate list. After the compact phase, this list is appended to the front of the existing wrong-way pointer list. In order to prevent duplicates in the wrong-way pointer list, objects known to contain wrong-way pointers are tagged when first encountered in a store list. Subsequent store list elements pointing to such tagged objects are discarded.

After the mark phase of a major collection, the mark bits are used as a source of heap density information. At the start of the compact phase, the mark bits are scanned but the heap is not compacted until the cumulative density drops below a preset *density threshold*. After the density threshold is reached, the remainder of the heap is compacted.

PERFORMANCE EVALUATION

Methodology

Appel[20] used six benchmark programs to evaluate the performance and code quality of the SML-NJ compiler. Our performance evaluation is based on the same six programs, with one additional program, which we shall describe shortly. Two aspects of performance were of primary interest: the effect of the density heuristic in reducing the movement of long-lived objects, and the impact of wrong-way pointers.

Initial experience with the original six programs revealed that the programs roughly fell into three classes:

1. The programs Life, Lex and Knuth–Bendix do not allocate many long-lived objects and do not exhibit a significant number of major garbage collections.
2. The programs VLIW, Yacc and the ML compiler itself all create a significant number of long-lived objects and undergo multiple major garbage collections. All three programs make extensive use of both structured data and the assignment statement.
3. The program Simple is in a class of its own. This program is a manual translation of a spherical fluid-dynamics program originally written in FORTRAN. The program is characterized by its repeated assignments into large fixed arrays. Like the programs in the first class, Simple exhibits relatively few major collections compared with the programs in the second class, but it is of special interest to us because of its use of assignment and the consequent creation of wrong-way pointers.

Since the relative performance of our garbage collector was likely to be the most impressive with a purely-functional program, in which no wrong-way pointers would be created, a

synthetic, purely-functional program was created. This synthetic program, which we named 'Funct' performs a variety of typical data manipulation tasks: (1) generates a list of 125000 random integers; (2) inserts them into a binary tree; (3) looks up each element of the original list in the tree; (4) traverses the tree and extracts an in-order list; (5) verifies the order of the in-order list; (6) looks up each element of the in-order list in the tree; (7) uses quicksort to produce another ordered list of the integers; and (8) compares the ordered list from the quicksort to the ordered list from the tree traversal.

In view of our initial experience, we decided to base our performance evaluation on three benchmarks:

1. Two executions of Funct. Execution of the benchmark requires an heap of 15–22MB with an overall image size of 30–40MB. During execution the benchmark executes 15 or more major collections, depending on the density threshold. After each major garbage collection about 9–14MB of data remains live. On a Sun 4/670, with sufficient physical memory available to prevent paging, it runs for about three minutes.
2. Execution of the compiler compiling Lexgen, Knuth-Bendix, Simple, Denote, and Funct (the 'Compile' benchmark). Execution of the benchmark uses an heap of 15MB with an overall image size of 18–28MB. During execution the benchmark performs nine or more major collections. After each major collection about 9MB of data remains live. On a Sun 4/670, with sufficient physical memory available to prevent paging, it runs for about two minutes.
3. Two executions of Simple, the 'most imperative' program available. It uses an heap of 11MB with an overall image size of 18–24MB. During execution the benchmark performs two or more major collections. After each major collection about 8MB of data remains live. On a Sun 4/670, with sufficient physical memory available to prevent paging, it runs for just over 1·5 minutes.

All of the benchmarks perform hundreds of minor collections during each run.

Because of the similarity of the Yacc and VLIW programs to the Compile benchmark, and to keep benchmarking time reasonable, we removed them from the benchmark suite. We felt that this set of benchmarks would provide adequate breadth and test the garbage collector under both the least and most favorable circumstances.

Benchmarking was performed on a Sun 4/670 running version 4.1.2 of SunOS. The machine uses a two-level cache. The first level consists of a 20KB five-way set associative instruction cache with a 8 byte line size and a 16KB write-through four-way set associative data cache with a 4 byte line size. At the second level is a combined 1MB direct-mapped copy-back cache with a 128 byte line size.

During benchmarking the system was otherwise unloaded except for essential daemons (init, swapper, pagedaemon) and the processes supervising the benchmarking. In order to evaluate virtual memory performance, physical memory available to the benchmarks was restricted. This restriction of physical memory was achieved by having one of the supervisory processes lock the unwanted pages of physical memory. It is difficult to determine exactly the amount of memory available to the benchmark programs, as SunOS uses a common pool of pages for paging and file system caches. However, the benchmarks were run with (approximate) memory sizes from 10–35MB in 5MB steps. The benchmarks of our algorithm were executed with heap density thresholds between 60 and 100 per cent in 10 per cent increments. A single run of the complete benchmark suite, with six different memory sizes and six different density thresholds, takes about seven hours on a dedicated Sun 4/670.

Results

Figures 3–8 show our results at a range of memory sizes. All graphs show our collector at the various heap density thresholds represented by different line styles and the original SML-NJ collector represented by diamonds. All graphs are calculated as arithmetic means of three or more runs.

As with most performance evaluations, the total elapsed time seen by the user is of greatest interest. Elapsed time for our three benchmarks are shown in Figures 3, 4 and 5.

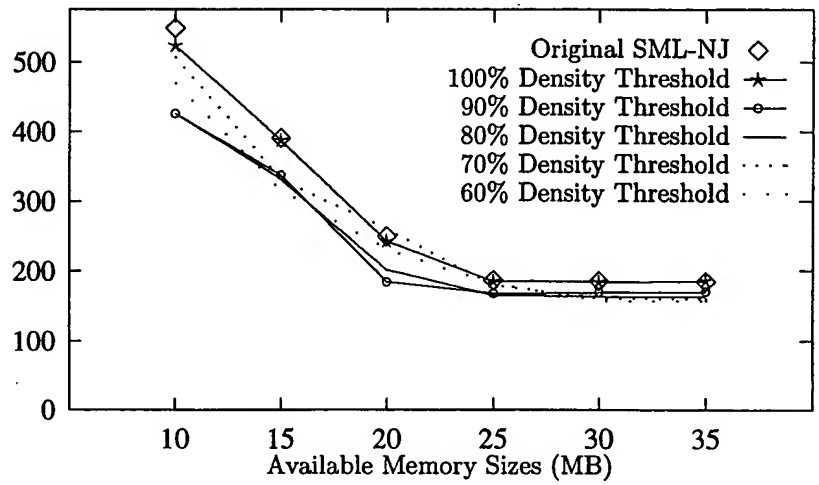

Figure 3. Total elapsed time (seconds) for Funct

Total elapsed times for Funct (Figure 3) show the most dramatic improvements. When only 10MB of physical memory was available, the program ran 23 per cent faster with our garbage collector (with a density threshold of 90 per cent) than with the original collector. When 35MB of physical memory was available, the improvement was 8 per cent.

Total elapsed times for compilation (Figure 4) also show reasonable improvement over the original collector. With a density threshold of 90 per cent, this improvement ranges from 19 per cent faster with 10MB of available physical memory to 3 per cent faster with 35MB of available physical memory.

The large number of wrong-way pointers in the heap affects the total elapsed times for Simple (Figure 5). With a density threshold of 90 per cent, we show a 6 per cent improvement with 10MB of available physical memory. With 35MB of available physical memory, the original garbage collector runs 2·4 per cent faster than our garbage collector with a 90 per cent density threshold.

For this last benchmark, the relatively poor performances of our garbage collector at the 80 per cent density threshold is surprising, although it is consistent over the three runs. The elapsed time difference is completely accounted for by mutator CPU time, but we have not yet been able to explain why the CPU time should be longer. Cache conflicts in the second-level direct-mapped cache may be responsible.

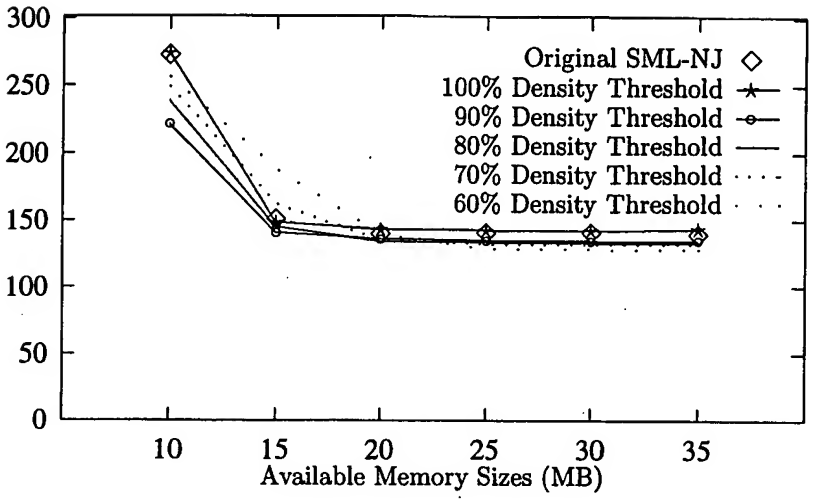

Figure 4. Total elapsed time (seconds) for compile

In general, we have found that a density threshold of 90 per cent seems to produce the best overall performance improvement. Density thresholds of 50–70 per cent reduce noticeably the amount of work the garbage collector has to do, but at the expense of virtual memory locality. A density threshold of 100 per cent maximizes virtual memory locality but requires more work on the part of the garbage collector. These locality effects have impact on both

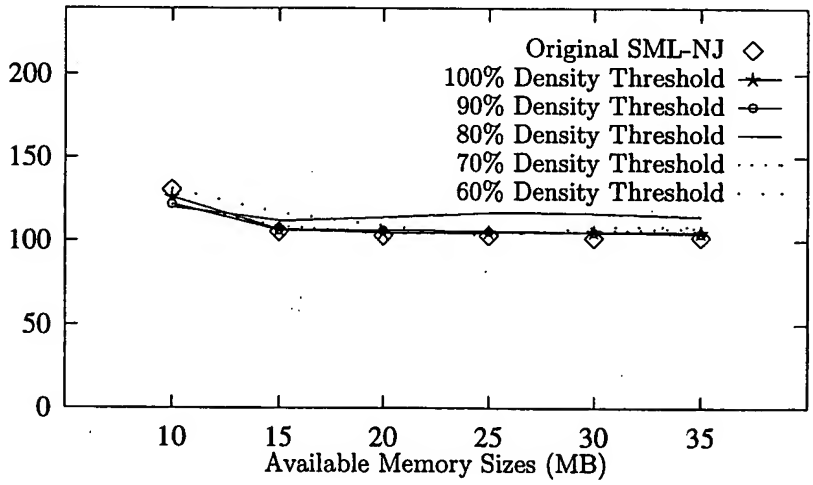

Figure 5. Total elapsed time (seconds) for simple

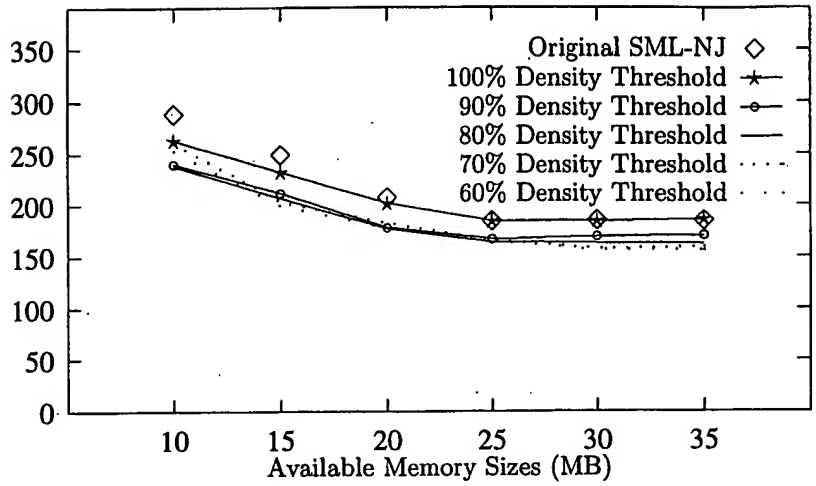

Figure 6. Total CPU time (seconds) for Funct

the collector itself and the mutator.

We now turn to comparing the factors that contribute to the total elapsed times. Total CPU time for Funct is plotted in Figure 6. At all density thresholds except 100 per cent the total CPU time using our collector is significantly less than the total CPU time when the original collector is used. Most of this CPU time improvement is due to improved garbage collector performance rather than from improved mutator performance. These times include locality effects on virtual memory performance from improved page fault rates during garbage collection. Total CPU time for other benchmarks shows similar, though less pronounced, trends.

The number of wrong-way pointers has a major effect on the CPU performance of the garbage collector. To illustrate this most strongly, we look at garbage collector CPU performance at the 100 per cent density level with 35MB of physical memory available. This is the situation that is least favorable to our garbage collector. The benchmarks exhibit few if any page faults and no benefit is obtained from the density heuristic, as both collectors move all objects during a major collection. In this situation the original collector is faster for all benchmarks. The relative speed is directly related to mutator use of the assignment statement. For the Funct benchmark the relative speed difference is 3·5 per cent. At the other extreme, it is 43 per cent for the Simple benchmark. Between these is the Compile benchmark at 17 per cent.

Total page faults for Funct are shown in Figure 7. Trends are similar to those seen for total CPU time. Most of the improved page fault rate is due to an improved page fault rate in the garbage collector rather than in the mutator. A claim is often made that copying collectors should exhibit better paging performance than mark-and-sweep collectors, since they look only at live data. In our case, garbage appears to be sufficiently intermingled with live data that few pages contain only garbage.

We have found limited support for the locality assumption. Figure 8 shows mutator elapsed

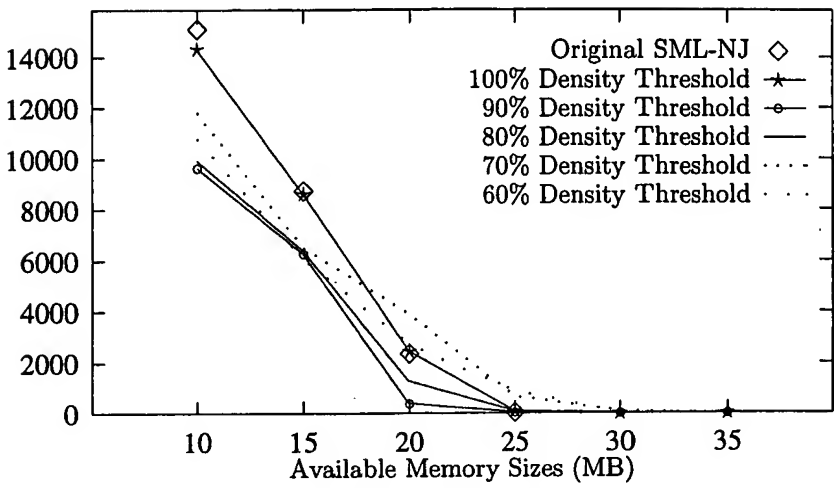

Figure 7. Total page faults for Funct

times for Funct. The garbage collector can affect these times only by improving locality and by the working set present at the end of a garbage collection. Low densities clearly have a negative effect on mutator performance. At higher densities, mutator performance is improved by as much as 13 per cent, but with 5 per cent being more typical.

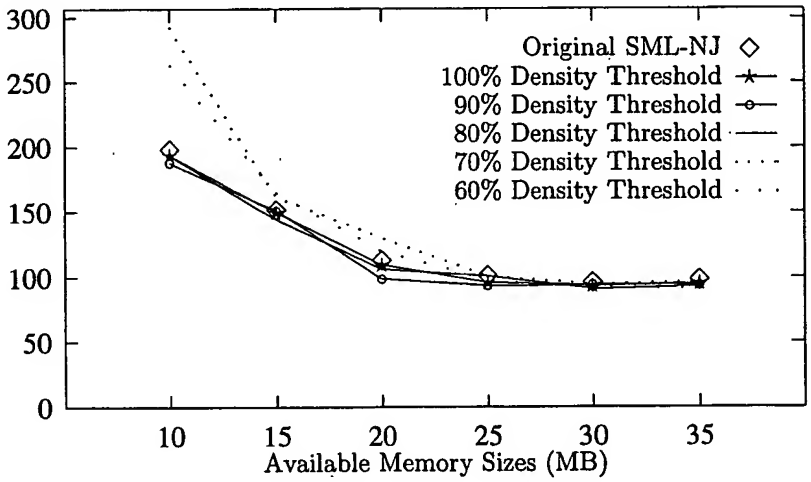

Figure 8. Mutator elapsed time (seconds) for Funct

RELATED WORK

Knuth[21] describes a simple compacting algorithm (the LISP 2 garbage collector). This collection algorithm requires an extra pointer field in each object, which the garbage collector uses to hold the address to which the object will be moved. After the mark phase, the sweep phase proceeds in three linear passes through memory: the first pass calculates the new location for each object, the second pass updates all references to point to the new locations, and the final pass moves the objects.

Other published algorithms for compacting garbage collection may be divided into *threading* collectors (those of Fisher,[22],Thorelli [23] Dewar and McCann,[24] Hanson,[25] Morris,[26,27] Jonkers[28] and Martin[29]) and *break table* collectors (those of Haddon and Waite,[30] Wegbreit,[31] Zave,[32] Fitch and Norman[33] and Terashima and Goto.[34]) Threading collectors use a technique in which the cells pointing at a given location are linked into a list headed by the referenced cell. When the new address for the referenced cell is determined, this linked list is traversed and the original structure is recreated with cells pointing at the new location. The break table collectors construct a data structure (the break table) holding relocation information and then use this data structure to compute relocation offsets for pointers. Storage for the break table is obtained by reusing locations formerly occupied by garbage. Our collector is essentially a break table collector with auxiliary storage for the break table. All of these collectors are similar in placing emphasis on the minimal use of storage beyond that required for the heap. Typically these algorithms have storage requirements of a few bits per object plus a small, constant amount of additional storage. In contrast, we are more concerned with virtual memory performance, and the effect of the garbage collection algorithm on total execution time of real programs.

Several authors have recognized the possible benefits of preserving allocation-order during garbage collection and the increased effectiveness of these algorithms if the ordering and persistence assumptions hold. Terashima and Goto[34] classify several garbage collection algorithms according to whether or not they preserve allocation order ('genetic order' in their terminology). The collector of Martin[29] has reduced storage requirements if all pointers to pointers run in the same direction. Fisher[22] describes a garbage collector for use in environments in which all pointers point in a single direction. The operation of his collector depends crucially on the allocation-order preserving property.

Hanson[25] anticipated generational garbage collection by several years in his description of a compacting garbage collector for a SNOBOL4 implementation. He notes that some objects are allocated early in the execution of a program and never become garbage. The collector assumes that these objects, called the 'sediment', lie below the first garbage cell found after each collection. The 'sedimentary floor' increases as the program executes. Objects below this floor are not subject to reclamation until an allocation request is made that cannot be satisfied without extending the arena size.

Like the tenuring in generational scavenging,[35] our algorithm tends not to move long-lived objects. Unlike tenuring, however, our algorithm *will* collect if enough of the long-lived objects become garbage. Other systems use a non-compacting mark-and-sweep collector for the oldest generation. Here too, long-lived objects are not moved when they reach the oldest generation. Storage that is reclaimed by the non-compacting mark-and-sweep collector is reused by objects being promoted into the oldest generation. Cheney's algorithm cannot be used to copy objects into this generation. An auxiliary data structure must be used to guide traversal of the objects being promoted and storage for each object must be allocated from a free list. As with all memory-management strategies where objects are not moved, this

approach can suffer from fragmentation problems and from locality problems, as objects with disparate ages become intermingled.[3] Concurrent Caml Light, an implementation of ML that supports threads, is an example of a system that uses this approach.[36].

There have been several attempts to find better ways to traverse the accessibility graph in order to improve locality.[37] Wilson, Lam and Moher[16] thoroughly examine the breadth-first ordering introduced by Cheney's algorithm,[8] the depth-first ordering described by Stamos,[15] as well as proposing and analyzing their own static clustering technique. They also report a comment by Andre[38] that Symbolics, Inc. found the original creation order of code to be much superior in locality to anything else they tried. This is the same order as our collector produces for code and data structures. Courts[14] describes a hardware solution to copy objects in the order that the program uses them.

Zorn[10] has demonstrated through a simulation study that mark-and-sweep collectors can exhibit performance comparable to that of copying collectors. His experiments were based on a mark-and-sweep collector that did not perform compaction: Objects are moved only when being promoted to an older generation.

Fernández and Hanson report experience with compacting and copying garbage collectors for the Icon programming language.[39] They replaced an existing compacting collector with a copying collector in an implementation of Icon. They discovered that, while the introduction of the copying collector produced improved performance, improvements could be made to the existing compacting collector that resulted in the best performance. The garbage collectors in question were not generational and virtual memory effects were not examined. Nonetheless, we take their results as highly supportive of the conclusions of the present paper.

O'Toole, Nettles and Gifford[40] describe a variant of copying garbage collection for the concurrent maintenance of a persistent heap. They use the equivalent of our forwarding array to avoid modifying objects in the active semi-space while the objects are being copied into the fallow semi-space. The use of the forwarding array was found to substantially simplify the design of the collection algorithm.

CONCLUDING DISCUSSION

For the purposes of our experiments, we made no changes to the portion of the SML-NJ run-time system that determines when and how the garbage collector is called. Determination of when the heap size should be increased or decreased, or when a major collection should be performed is made using the same scheme used by the original collector. Additional performance improvements might be expected if the run-time system were designed to complement a compacting garbage collector. Of particular interest is the possibility of having the density threshold adjusted automatically using feedback from system performance. Dealing with interior pointers was a major frustration that no doubt had a detrimental impact on performance. Since a compacting collector does not move objects relative to one another unless an intervening object becomes garbage, cooperation from the run-time system would obviate the need for special handling of interior pointers.

As an extension of this last observation, we are considering the benefits of the exclusive use of relative pointers by the run-time system. If relative pointers are used exclusively, any contiguous block of live objects may be treated as a group for garbage collection purposes. Such a group could be slid through memory without making adjustments to pointers interior to the group. A version of the density heuristic may be applicable in managing such groups, as it may not be of benefit to compact a group to reclaim a small amount of interior garbage.

These advantages of relative pointers might be outweighed by the additional work required for dereference. In order to dereference a relative pointer, the value of the pointer must be added to the address of its storage location before the object it points to may be accessed. An experimental study would be required to determine the practicality and benefits of this idea.

The garbage collection approach described in this paper causes the mutator to temporarily cease operation while a collection occurs. In real-time or interactive applications this delay cannot be tolerated and necessitates the collection of garbage incrementally, possibly at the expense of overall performance. Concurrent and incremental garbage collectors allow the operation of the mutator to be interleaved with the operation of the collector, avoiding long delays. Many people have described concurrent and incremental collectors.[3] Baker's algorithm[41] – an incremental copying collector based on Cheney's algorithm – is one early and well-known example. Concurrent and incremental algorithms have had the reputation of requiring significant support from the operating system and/or hardware to be efficiently implemented. However, concurrent collectors that require only mild virtual memory support have been recently developed in both copying[42] and non-compacting mark-and-sweep[43] flavors.

We are presently considering the implementation of a concurrent compacting collector based on our techniques. The mark phase of such a collector could be implemented using techniques borrowed from non-compacting concurrent mark-and-sweep collectors. On the other hand, the compacting phase causes some difficulty, as objects are moved within a single arena and it is difficult to determine whether a mutator reference is to the object occupying a location before compaction or to the object occupying it after compaction. It is possible to avoid this problem by compacting to a separate memory area that is then re-mapped through virtual memory back into the original location at the end of the garbage collection cycle. During a collection the mutator would continue to access the uncollected arena. Immediately before re-mapping, the store list could be used to update locations modified by the mutator during the collection. Similar techniques have been used in an experimental incremental copying collector for SML-NJ.[44] An alternative to this approach is to require the mutator to explicitly check on each reference for a garbage collection in progress and use the forwarding array as necessary to find an object's current location.

The effectiveness of our density heuristic is perhaps the most interesting result of the experiment. The majority of the performance improvement is due to the migration of long-lived objects to one end of the heap where they are rarely moved. The density heuristic is enabled by the allocation-order preserving properties of compaction. Equivalent heuristics are more difficult to incorporate into copying collectors. Performance of the garbage collector is best when few or no pointer assignments are made, as in a purely functional program. This observation leads to the conclusion that compacting collection is especially suited to pure functional programming languages. A reduced need to move data also suggests that compaction might be an appropriate choice for persistent data structures,[40,45,46] where moving an object involves a disk read and write.

In summary, a compacting collector should be considered a viable choice when designing the memory-management subsystem for a modern program development environment. Generational garbage collection is considered key to the performance of such subsystems. To our knowledge, the present paper represents the first time that a compacting generational garbage collector has been the subject of significant study. Compacting garbage collection and generational garbage collection are intimately related, supporting one another in beneficial ways.

ACKNOWLEDGEMENTS

Gord Cormack carefully read both preliminary and penultimate drafts of this paper and made numerous useful comments. Discussions with Gord Vreugdenhil and John Ophel benefited this work substantially. Naji Mouawad answered an early data structuring question; Peter Buhr suggested the technique of locking pages to restrict available memory; Lal George at AT&T provided us with the SML-NJ benchmark programs. The anonymous referees made several suggestions that increased the clarity and completeness of the exposition. Finally, we thank Andrew Appel for instigating the SML-NJ run-time system and documenting it well enough that we were able to replace his garbage collector. The first author was funded by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

1. Jacques Cohen, 'Garbage collection of linked data strucutures', *Computing Surveys*, **13**(3), 341–367 (1981).
2. Andrew W. Appel, 'Garbage collection', in P. Lee (ed.), *Advanced Language Implementations*, MIT Press, 1991, chapter 4, pp. 89–100.
3. Paul R. Wilson, 'Uniprocessor garbage collection techniques', Yves Bekkers and Jacques Cohen (eds.), *Memory Management – International Workshop IWMM 92*, St. Malo, France, September 1992, pp. 1–42. Published as *Springer Lecture Notes in Computer Science, 637*, Springer-Verlag, Berlin.
4. Andrew W. Appel, 'Garbage collection can be faster than stack allocation', *Information Processing Letters*, **24**(2) (1987).
5. Andrew W. Appel, 'Simple generational garbage collection and fast allocation', *Software–Practice and Experience*, **19**(2), 171–183 (1989).
6. Jacques Cohen and Alexandru Nicolau, 'Comparison of compacting algorithms for garbage collection', *ACM Transactions on Programming Languages and Systems*, **5**(4), 532–553 (1983).
7. Robert R. Fenichel and Jerome C. Yochelson, 'A LISP garbage-collector for virtual-memory conputer systems', *Communications of the ACM*, **12**(11), 611–612 (1969).
8. C. J. Cheney, 'A nonrecursive list compacting algorithm', *Communications of the ACM*, **13**(11), 677–678 (1970).
9. Hans-Juergen Boehm and Mark Weiser, 'Garbage collection in an uncooperative environment', *Software – Practice and Experience*, **18**(9), 807–820 (1988).
10. Benjamin Zorn, 'Comparing mark-and-sweep and stop-and-copy garbage collection', *ACM Symposium on LISP and Functional Programming*, 1990, pp. 87–98.
11. Henry Lieberman and Carl Hewitt, 'A real-time garbage collector based on the lifetimes of objects', *Communications of the ACM*, **26**(6), 419–429 (1983).
12. Douglas W. Clark and C. Cordell Green, 'An empirical study of list structure in Lisp', *Communications of the ACM*, **20**(2), 78–87 (1977).
13. Douglas W. Clark, 'Measurements of dynamic list structure use in Lisp', *IEEE Transaction on Software Engineering*, **5**(1), 51–59 (1979).
14. Robert Courts, 'Improving locality of reference in a garbage-collecting memory management system', *Communications of the ACM*, **31**(9), 1128–1138 (1988).
15. James W. Stamos, 'Static grouping of small objects to enhance performance of a paged virtual memory', *ACM Transactions on Programming Languages and Systems*, **2**(2), 155–180 (1984).
16. Paul R. Wilson, Michael S. Lam and Thomas G. Moher, 'Effective "static-graph" reorganization to improve locality in garbage-collected systems', *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991, pp. 177–191. Toronto, Canada.
17. Andrew W. Appel and David B. MacQueen, 'Standard ML reference manual'. (Preliminary), October 1990.
18. Robin Milner, Mads Tofte and Robert Harper, *The Definition of Standard ML*, MIT Press, Cambridge, Massachusetts, 1990.
19. Andrew W. Appel, 'A runtime system', *LISP and Symbolic Computation*, **3** (1990).
20. Andrew W. Appel, *Compiling with Continuations*, Cambridge University Press, 1992.
21. Donald E. Knuth, *The Art of Computer Programming*, Volume 1, pp. 454, 602–603, Addison-Wesley, Reading, Massachusetts, 1968.
22. David A. Fisher, 'Bounded workspace garbage collection in an address-order preserving list processing

environment', *Information Processing Letters*, **3**(1), 29–32 (1974).
23. Lars-Erik Thorelli, 'A fast compactifying garbage collector', *BIT*, **16**(4), 426–441 (1976).
24. R. B. K. Dewar and A. P. McCann, 'MACRO SPITBOL – A SNOBOL4 compiler', *Software – Practice and Experience*, **7**(1), 95–113 (1976).
25. David R. Hanson, 'Storage management for an implementation of SNOBOL4', *Software – Practice and Experience*, **7**, 179–192 (1977).
26. F. Lockwood Morris, 'A time- and space-efficient garbage collection algorithm', *Communications of the ACM*, **21**(8), 662–665 (1978).
27. F. Lockwood Morris, 'Another compacting garbage collector', *Information Processing Letters*, **15**(4), 139–142 (1982).
28. H. B. M. Jonkers, 'A fast garbage compaction algorithm', *Information Processing Letters*, **9**(1), 26–30 (1979).
29. Johannes J. Martin, 'An efficient garbage compaction algorithm', *Comunications of the ACM*, **25**(8), 571–581 (1982).
30. B. K. Haddon and W. M. Waite, 'A compaction procedure for variable-length storage elements', *The Computer Journal*, **10**, 162–165 (1967).
31. B. Wegbreit, 'A generalized compactifying garbage collector', *The Computer Journal*, **15**(3), 204–208 (1972).
32. Derek A. Zave, 'A fast compacting garbage collector', *Infomation Processing Letters*, **3**(6), 167–169 (1975).
33. J. P. Fitch and A. C. Norman, 'A note on compacting garbage collection', *The Computer Journal*, **21**(1) (1978).
34. Motoaki Terashima and Eiichi Goto, 'Genetic order and compactifying garbage collectors', *Information Processing Letters*, **7**(1), 27–32 (1978).
35. David M. Ungar, 'Generation scavenging: A non-disruptive high-performance storage reclamation algorithm', *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, pp. 157–167. Also published as *ACM SIGPLAN Notices*, **19**(5), 157–167, May, 1987.
36. Damien Doligez and Xavier Leroy, 'A concurrent, generational garbage collector for a multithreaded implementation of ML', *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993, pp. 113–123.
37. Paul R. Wilson, 'Some issues and strategies in heap management and memory hierarchies', *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990. Also published as *SIGPLAN Notices*, **23**(1), 45–52, January 1991.
38. David L. Andre, 'Paging in LISP programs', *Master's Thesis*, University of Maryland, 1986.
39. Mary F. Fernandez and David R. Hanson, 'Garbage collection alternatives for icon', *Software – Practice and Experience*, **22**(8), 659–672 (1992).
40. James O'Toole, Scott Nettles, and David Gifford, 'Concurrent compacting garbage collection of a persistent heap', *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, December 1993, pp. 161–174. Also published as *Operating Systems Review*, **27**(5), 161–174, December, 1993.
41. Henry G. Baker, Jr, 'List processing in real time on a serial computer', *Communications of the ACM*, **21**(4), 280–294 (1978).
42. Andrew W. Appel, John R. Ellis and Kai Li, 'Real-time concurrent collection on stock multiprocessors', *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988, pp. 11–20.
43. Hans-J. Boehm, Alan J. Demers and Scott Shenker, 'Mostly parallel garbage collection', *Programming Language Design and Implementation*, 1991, pp. 157–164.
44. Scott Nettles, James O'Toole, David Pierce and Nicholas Haines, 'Replication-based incremental copying collection', Yves Bekkers and Jacques Cohen (eds.), *Memory Management – International Workshop IWMM 92*, St. Malo, France, September 1992, pp. 357–364. Published as *Springer Lecture Notes in Computer Science, 637*, Springer-Verlag, Berlin.
45. Elliot K. Kolodner and William E. Weihl, 'Atomic incremental garbage collection', Yves Bekkers and Jacques Cohen (eds.), *Memory Management – International Workshop IWMM 92*, St. Malo, France, September 1992, pp. 365–387. Published as *Springer Lecture Notes in Computer Science, 637*, Springer-Verlag, Berlin.
46. Elliot K. Kolodner and William E. Weihl, 'Atomic incremental garbage collection for a large stable heap', *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 1993, pp. 177–186. Also published as *ACM SIGMOD Record*, **22**(2), 177–186, June, 1993.